

LINUX EXPERT

FAQ

Q I want to sell my computer, but I'm frightened someone will be able to recover personal files. Can I remove them for good?

A A growing area in identity theft stems from the sale of old hardware. Because the command line has no Recycle Bin, many users and administrators believe that what they delete with the `rm` command is gone for good. However, there are several clever utilities that simply undelete files by reconstituting them from raw data found on the disk.

Instead of using `rm`, try `shred`. Rather than simply marking the disk space as reusable, `shred` will overwrite the file a default 25 times, first of all with random data and then with 24 other patterns:

```
# shred myfile.txt
```

The `-v` switch will show you the patterns `shred` uses to overwrite the file. You can ensure that `shred` finally overwrites the file with zeros to hide its activities with the `-z` switch, too. The `-u` switch will also truncate the file to zero bytes.

Jon Thompson
UNIX/Linux management
and security consultant

linux@computershopper.co.uk

Regular Expressions

Whatever you're looking for in Linux, regex can help you hunt it down with ease and precision. Jon Thompson explains how to use this versatile tool for all kinds of search patterns

Your Linux distribution includes a rich set of data-manipulation tools, including over 100 you can use to create powerful commands and scripts. Knowing a bit about these utilities and their capabilities will allow you to manipulate your system and its data easily, in ways that are too difficult to achieve without a lot of organisation in other operating systems.

One key way of managing and manipulating your system is searching, either for files and directories, or for content within files. It may be that you simply need to find the location of a piece of text or want to extract information from a log file according to some special criteria and process it further. You may want to pull out certain events from your firewall log to gauge whether you're attracting the attentions of a hacker, or search a configuration file for a troublesome directive. You might need to run a more complicated search than for a simple string of characters, and find the usual asterisk and question mark wildcards can't achieve pin-point accuracy on their own. If any of these apply to you, this month's *Linux Expert* will help.

EXPRESS DELIVERY

Regex (Regular Expressions) provides a highly flexible and expressive way of creating very precise search patterns. Unfortunately, regex has developed a bad reputation since its introduction into early UNIX systems because of its impenetrable documentation. If you have heard of regex and tried to use it before, you probably gave up in frustration shortly after starting. The documentation for regex seems to be written in Martian, and a quick survey of online tutorials shows that many of them are written with the professional system administrator or programmer in mind, rather than more casual users.

Once you understand the basic principles, however, it is very easy to use regex to build search patterns to find whatever you need. We'll take you on a tour of the basics and show you how to combine them into something that goes way beyond simple wildcards. We've included a handy table, listing the function of all the parts of regex we'll study, and we've littered the text with examples to help you when you come to build your own expressions.

Regex is all about searching, so first we need a substantial piece of text on which to practise. Let's generate some by copying the contents of the `coreutils` documentation into a text file as follows:

```
# info coreutils > text.txt
# wc text.txt
14525 82757 625868 text.txt
```

The `wc` (word count) command gives us some statistics about the file we've generated. It's over 14,000 lines long, with more than 82,000 words and over half a million characters. If you browse the file using the more command, you will be able to see that we have plenty

of text with which to explore the capabilities of regex-aware commands.

Let's jump right in by using the `egrep` command to find patterns in `text.txt`. We use the `egrep` command, as opposed to the more traditional `grep`, because it allows us to do some useful things we cannot do with standard `grep`. First, we ask it to match lines containing the simple pattern `Ab`:

```
# egrep "Ab" text.txt
* Date conversion specifiers::  %[aAbBcCdD
eFgGhJmUvWwXyY]
* Date conversion specifiers::  %[aAbBcCdD
eFgGhJmUvWwXyY]
```

`Egrep` has found just two lines that match the input pattern. The pattern itself can appear anywhere in the line, either on its own or as part of a longer string of characters, as it does above.

There's also a useful character we can include in our search pattern. The full stop (`.`) allows us to specify a wildcard for a single unknown letter. For instance:

```
# egrep "Ab.c" text.txt
* Date conversion specifiers::  %[aAbBcCdD
eFgGhJmUvWwXyY]
* Date conversion specifiers::  %[aAbBcCdD
eFgGhJmUvWwXyY]
```

`Egrep` still returns the two lines as before, but we've now extended the pattern to include a trailing `c` and told it to accept any character between it and `b`. In both cases, the use of the regex wildcard matched a `B`.

We can easily use regex to tell `egrep` to match a pattern only if it appears at the start or end of the line. The following displays lines that begin with the letter `T`:

```
# egrep "^T" text.txt | more
This manual documents version 5.96 of the
GNU core utilities, including
This manual is a work in progress: many
sections make no attempt to
The 'cp', 'install', 'ln', and 'mv'
commands normally treat the last
```

The full output continues with over 80 further lines of text, all beginning with a capital `T`. We've used a combination of the pipe command and the `more` command to display one page at a time. The `^` (carat) character is regex for 'match the following pattern of letters, but only if it appears at the beginning of a line'.

CHARACTER DEVELOPMENT

The letters that make up a search pattern are called literals, but there are also special characters we can include that give us more control over how we search for those literals. Combined, these are called expressions. The caret symbol is one such control character, as is the period. These and the other control



characters are called metacharacters, and there are quite a few. See the table on page 185 for the most useful ones.

We can, for instance, indicate that the pattern should be matched only if it occupies the end of a line by placing a dollar metacharacter immediately after it:

```
# egrep ")$" text.txt | more
```

This expression makes egrep output all lines that end in a closing bracket. This use of metacharacters to ensure the search hit is at the start or end of a line is called anchoring. You can combine the two anchors into a regex expression to ensure that the pattern is the only thing on the line. For example:

```
# echo "Hello" | egrep "^Hello$"
Hello
```

If we want to search for a caret, dollar, full stop or other character that is used as a metacharacter, we have to place a backslash before the character. This causes it to be treated as a literal and not a metacharacter:

```
# echo "^Hello" | egrep "\\^Hello"
^Hello
```

This technique is called escaping. Similarly, if you want to search for a backslash, simply place another backslash before it. For example:

```
# echo "\\Hello" | egrep "\\Hello"
\\Hello
```

Searching for strings of literals is OK, but we can increase the power of our regex expressions by creating those that search for variations on a pattern, too.

THIS AND THAT

Suppose we want to search for lines that begin with This or That? We could search the file twice, appending the second search results to the first, as follows:

```
# egrep "^This" text.txt > out.txt
# egrep "^That" text.txt >> out.txt
```

The problem here is that the lines returned by these commands don't end up listed in the out.txt file in the same order as they appear in the original file. All the This lines will appear before the That lines. This could get messy, particularly if you are dealing with log files where you want to keep the time-stamped lines in the correct order. There's a better way to extract these two patterns, which introduces a flexible feature of regex:

```
# egrep "^Th[ai]" text.txt | more
```

The square brackets are another example of metacharacters. They enclose a range of alternative characters that egrep can choose from in matching single literals in the pattern. This expression tells egrep to return all lines beginning with 'Th' and followed immediately by either an 'a' or an 'i'. The expression will match lines beginning with This and That, but not The. We can include this third option as follows:

```
# egrep "^Th[ai]" text.txt | more
```

Square brackets can also be used to specify negatives, which are characters you don't want to match. For example:

```
# egrep "^Th[^e]" text.txt | more
```

This will match all lines beginning with Th except those beginning with The. When placed directly after an opening square bracket, the carat means 'but not the following'.

RANGE ROVERS

Bracketed alternative literals are expressive on their own, but if you have a few alternatives (such as the digits 1, 4, 9 and a selection of alphabetic characters), creating such lists can be tedious or prone to error. Luckily, regex provides an efficient way around this. Square brackets can contain ranges of literals to match. If we wanted to match lines containing at least a single digit, we could use what we've learned so far to provide an expression like this:

```
# egrep "[0123456789]" text.txt | more
```

Using a range, however, we can shorten this command to the following:

```
# egrep "[0-9]" text.txt | more
```

This works with ranges of letters, too:

```
# egrep "[a-z]" text.txt | more
```

That last command will hunt for any lines beginning with a lower-case letter, while the following will return only lines beginning with upper-case letters:

```
# egrep "[A-Z]" text.txt | more
```

We can combine the two by separating the ranges with a comma, which is a metacharacter:

```
# egrep "[A-Z,a-z]" text.txt | more
```

There are some useful predefined ranges:

RANGE	MEANING
[:alnum]	Match any digit or letter
[:alpha]	Match any letter
[:cntrl]	Match a control character
[:digit]	Match a digit
[:graph]	Match any printable character
[:lower]	Match only a lower-case letter
[:print]	Match printable letters (including any spaces)
[:punct]	Match punctuation marks (£\$%^&*)
[:space]	Match a space
[:upper]	Match only an upper-case letter
[:xdigit]	Match a hexadecimal digit

In regex, these are known as character classes. As an example of how useful they can be, consider the following expression. It is a complex one, and it matches any upper-case or lower-case letter or a number (that is, any alphanumeric but not punctuation symbols):

```
# egrep "[A-Z,a-z,0-9]" text.txt | more
```

We can rewrite this in a far simpler way using the character class `:alnum:`, short for `alphanumeric`:

```
# egrep "[:alnum:]" text.txt | more
```

The second version is easier on the eyes. Some character classes are longer to type than specifying the range in longhand. For example, `[:lower:]` is longer than `[a-z]`, but in complex expressions using a character class can keep scripts more readable.

REPETITIVE STRESS

Simple bracketed lists of alternatives and ranges are great for use in patterns where we want to cover many possibilities for single literals, but what about situations where we don't know how many instances of a single character there are, or where we know there's a mix of characters in the middle of a pattern? These instances still require handling if we are to match patterns successfully and retrieve the proper data. Luckily, several metacharacters can help us.

To explain further, first consider the following text placed in a file called `abc.txt`:

```
ac
abc
abbc
abbbc
abbbbc
abbbbbc
abbbccc
```

If we wanted to find all the lines that contain several consecutive instances of the letter b, we could try:

```
# egrep "ab*c" abc.txt
```

This expression tells egrep to return all lines containing a pattern that starts with a, then zero or more instances of the literal b, and ends with a c.

Not surprisingly, this expression returns every line, including the first, because the asterisk matches zero or more instances. To return only those lines that contain one b or more, we would have to type:

```
# egrep "abb*c" abc.txt
abc
abbc
abbbc
abbbbc
abbbbbc
```

This expression returns lines containing a pattern beginning with ab followed by as many extra instances of b as there are to find, and ending in a c, but it's still not quite right because we want our expression to return only those lines that contain multiple instances of b. To do so, we can say:

```
# egrep "abbb*c" abc.txt
abbc
abbbc
abbbbc
abbbbbc
```

Here is an example of how to combine the asterisk with a list of alternatives to match in square brackets:

```
# egrep "b[bc]*" abc.txt
abc
abbc
abbbc
abbbbc
abbbbbc
```

This expression matches lines that contain a b followed by any number of 'b's or 'c's, including none at all. If we were to add the line abbbcccd to the file, we could extract it again using the expression:

```
# egrep "b[bc]*d" abc.txt
abbbcccd
```

This expression asks egrep to find patterns that start with a b, which then possibly have a mixture of b and c, and which then end with a trailing d. Suddenly, we've gone from explicitly giving the search pattern to letting egrep figure out valid combinations for itself. This is where the real power of regex lies.

The asterisk isn't the only regex metacharacter that can match a variable number of instances of preceding literals. There's also the plus metacharacter. This is an extended metacharacter, which is why we're using egrep rather than grep. Whereas an asterisk will match zero or more instances of a literal, a plus matches one or more instances, in this case b:

```
# egrep "b+" abc.txt
abc
abbc
abbbc
abbbbc
abbbbbc
abbbcccd
```

The question mark is another metacharacter we can use here, and it matches exactly zero or one instances of a literal. It's useful in cases where you're not sure whether a letter will appear or not, but want to cover all options. Here's an example:

```
# echo "colour" | egrep "colou?r"
colour
# echo "color" | egrep "colou?r"
color
```

It's the same pattern, but it matches both the UK and US spellings of 'colour' by making the instance of the letter u optional.

We can now match zero, one or many instances of a literal or group of alternatives. But what about matching a specific number of repetitions? Egrep will let you do this, too:

```
# egrep "ab{1,3}c" abc.txt
abc
abbc
abbbc
abbbcccd
```

Here we've specified that we want to match a pattern beginning with an a, followed by between one and three instances of b and ending in a c. If the first number in the curly brackets is lower than the second, then egrep will return an error:

```
# egrep "ab{3,1}c" long.txt
egrep: Invalid content of \{\}
```

FAQ Advanced searching

Q Can the find command use regex to do advanced searching?

A Indeed it can. Here's a typical find command, without regex, that looks for all files ending with .txt:

```
# find . -name *.txt -print
```

If you take out the -name switch and use the -regex switch instead, you can specify regex expressions for filenames, giving the command a lot more power. The following will find files in the current directory that begin with a number:

```
# find . -regex './[0-9].*' -print
```

Like grep and egrep, the find command can

also use regex without case sensitivity, using the -iregex switch. You can use extended regex by including the '-regextype posix-extended' switch:

```
# find . -regextype posix-extended
-regex './[0-9].*' -print
```

You can also use find to immediately invoke another command to run on the files it finds using the -exec switch:

```
# find . -regex './[0-9].*' -exec
rm -f {} \;
```

This command executes rm (to remove the files), representing the filenames it passes with the curly brackets. The end of the command is marked by the \;

As with other metacharacters, we can combine the plus, asterisk and question mark with square brackets. An expression such as [abc]* will, for instance, match all combinations of a, b, c, ab, ac and so on.

MULTIPLE CHOICE

So far, we've explored matching single literals. Extended regex also gives us the ability to match several alternative strings of literals. We specify these using parentheses. Here are a couple of quick examples that explain the concept:

```
# echo "my pet is a cat" | egrep
"(cat|dog)"
my pet is a cat
# echo "my pet is a dog" | egrep
"(cat|dog)"
my pet is a dog
```

We've expressed the choices of cat and dog inside parentheses, and separated them with the pipe symbol, which means 'or'. You can have as many choices as you like within the parentheses, and they don't have to be a single word, either:

```
# egrep "(This|That|Other)" text.txt
| more
# egrep "(This is|That was|The
other)" text.txt | more
```

You can also combine choices with other metacharacters. This example will return a match only if it appears at the start of a line:

```
# egrep "^ (This|That)" text.txt
| more
```

MARKING BOUNDARIES

So far, we've covered many of the abilities regex has when it comes to matching patterns and strings of literals, even those that are quite vague. We've also explored how to anchor those patterns to the start or end of the line.

So how do we find words? Placing a space before and after a pattern to mark it out as a complete word isn't always enough to catch all instances of it. The pattern might appear at the beginning of the line where there is no leading space, for instance. It might appear at the end of a line, or it might have a trailing comma or full stop. All these things will prevent us finding

every instance of it. Once again, regex provides us with a solution by recognising word boundaries. Here's an example:

```
# egrep "\bTh[ia][st]\b" text.txt
| more
```

The use of \b at the start and end of the expression marks the start and end boundaries of the word we're searching for. To the left and right must be white space (including a tab), a comma, a full stop or the beginning or end of a line. In other words, egrep is only to return matches that are full words and not parts of others. We can combine these boundaries with anchors, too, making a useful combination that looks at the first and last words on each line:

```
# egrep "^ \bTh[ia][st]\b" text.txt
| more
# egrep "\bTh[ia][st]\b$" text.txt
| more
```

Note the leading caret and trailing dollar signs anchoring each search. Here \b is an example of what's known as a shorthand character class. There are a few of these, all beginning with a backslash. Hopefully \n will be familiar to C and Perl programmers, as it matches any character considered to end a line, such as a new line or line feed. You can use it in combination with other characters or on its own, as follows:

```
# egrep "\n" text.txt
| more
```

This expression will return all lines containing a new line (which should be all of them). Meanwhile, \s will match a single instance of a space. In most implementations of regex, \s treats tab characters as spaces, too. However, \t will just match individual tabs.

As usual, you can mix shorthand character classes with other metacharacters. To search for instances of exactly four spaces, for example, you could use the expression \s{4,4}.

ALL TOGETHER NOW

Wouldn't it be great if we could match two words on a single line that appear near each other? Some search tools allow you to specify how many words apart the two expressions can be to make a match, and this is usually



configurable. Armed with regex, it's not too difficult to implement this yourself.

Our expression needs to match the first word, then some intermediate words and finally the second word. To help us, the `\w+` and `\W+` shorthand character classes will match words and punctuation characters respectively. We'll use `word1` and `word2` for the two words we wish to match, between one and six words apart.

The `\W+(?:\w+\W+)` part of the expression below looks complex, but it just tells regex to match one or more punctuation characters, then either a word or one or more punctuation characters. The `?:` is the key to this. It is a special operator that stops regex getting confused and trying to create matches by combining the `\W+` to the left of the brackets and the patterns inside the brackets. After all, we want it to match complete words only:

```
"\bword1\W+(?:\w+\W+){1,6}word2\b"
```

There are times, however, when there's no short cut. What do you do then?

You can create some extremely expressive and precise patterns using regex. One way is to nest the various metacharacters, as follows:

```
# egrep "(^bthe(re|ese)\b|There)"
text.txt | more
```

This expression looks complex, but it's actually very simple. It tells `egrep` to match a complete word at the start of every line beginning with 'the' and ending in either 're' or 'ese', and to match lines containing the pattern 'There' anywhere in their length.

Because you can combine regex metacharacters in this way, it's easy to get into a muddle. When building your own, it pays to start simply and add complexity to narrow down the text returned until it's exactly what you want. This is how we generated the above expression. We knew we wanted to match 'there' and 'these', and so we created an initial pattern:

```
"the(re|ese)"
```

We knew it had to be at the start of a line, and that we wanted to find only whole words, so the pattern became:

```
"^bthe(re|ese)\b"
```

We also knew that we wanted to find instances of `There`, so we added more parentheses to make two overall choices:

```
"(^bthe(re|ese)\b|There)"
```

Many beginners make the mistake of jumping in with `egrep` or some other regex-aware command and simply entering what they hope is the correct sequence of literals and metacharacters. This quickly leads to frustration when they are rewarded with the wrong output, an error or nothing at all. It's far better to start with a simple expression and refine it slowly until it does exactly what you need.

To help you construct your own expressions, and to illustrate the options available to you, we have supplied a handy reference guide to the metacharacters we've covered, including some simple examples of their use, at the bottom of this page.

It's not just commands such as `grep` and `egrep` that use regex. Linux programming languages such as Perl and PHP all have access to the regex system library and they can perform searching duties on files and data structures using regex expressions. See your programming documentation for more details.

HINTS AND TIPS

Your first introduction to regex can be slightly daunting, but you'll make life easier if you use a series of simple expressions and then pipe the output of one command into another. This means you can perform your processing in several easy steps. For instance, you might want to extract lines that contain the word 'error' but don't contain a number:

```
# egrep "error" logfile.txt | egrep
"^[0-9]"
```

This multi-step technique extends to other useful commands that don't use regex directly but whose output you can pipe into those that do. Here's an example of using regex to filter the output of the `ls` command to return all files that have the extension `.pl` or `.c`:

```
# ls -l | egrep "(\.pl$|\.c$)"
```

You can create complex if...then sequences using extended regex and a number of choices in parentheses. In the following example, we want `grep` to match all instances of 'read' and 'write' regardless of whether the initial letter is capitalised or not:

```
# egrep "([rR].ead|[wW].rite)"
text.txt
```

The `egrep` command itself has some useful command-line switches. If you include the `-A` switch followed by a number, it will return the lines containing your search pattern as normal, as well as some lines that follow on after it in the file. You specify how many of these lines you want using the number after the `-A` switch. Each set of lines of output is separated by a line containing `--`:

```
# egrep -A 2 "Hello" text.txt
Hello there.
This is a test of egrep's
-N facility
--
Hello again.
This also a test
of egrep's -A facility
```

This is particularly useful if you are scanning through log files. Not only can you find a line according to a pattern, you can also see any number of events that follow it. The opposite of the `-A` switch is `-B`. This prints out the specified number of lines that appear before each line containing the matched pattern.

Suppose you just want to know how many lines in a file match your search pattern. The `-c` switch built into `egrep` is used to count these lines:

```
# egrep -c "error" text.txt
249
```

You can also create a file containing multiple expressions and tell `egrep` to use them as a batch, instead of expressing them laboriously on numerous command lines. To do this, use the `-f` switch:

```
# egrep -f patterns.txt text.txt
```

Finally, if you need to match upper-case and lower-case patterns, you'll make life easier by telling `egrep` to ignore its usual Linux case sensitivity. To do this, simply use the `-i` switch and specify your patterns in lower case. **CS**

METACHARACTER	MEANING
<code>\</code>	Placed before a metacharacter to turn it into a literal; to match a full stop instead of using it as a single-character wildcard, you would use <code>\.</code>
<code>.</code>	Match a single character. So <code>ab.d</code> will match <code>abcd</code> and <code>abzd</code>
<code>?</code>	Match zero or exactly one instance of the character or expression immediately preceding the question mark. So <code>colou?r</code> matches <code>color</code> and <code>colour</code>
<code>*</code>	Match zero or more of the character or expression immediately preceding the asterisk. So <code>Sho*pper</code> will match <code>Shpper</code> , <code>Shopper</code> and <code>Shoopper</code>
<code>+</code>	Match at least one of the previous characters or expressions. So <code>Sho+pper</code> matches <code>Shopper</code> , <code>Shoopper</code> and <code>Shooopper</code>
<code>{x,y}</code>	Match between <code>x</code> and <code>y</code> instances of the character or expression immediately preceding the curly brackets. So <code>ab{1,3}c</code> will match <code>abc</code> , <code>abbc</code> and <code>abbbc</code>
<code>[]</code>	Match none or more of the characters in the brackets. So <code>a[bcd]</code> will match <code>ad</code> , <code>abd</code> and <code>acd</code>
<code>[^]</code>	Don't match any of the characters in the brackets. So <code>a[^b]c</code> will not match <code>abc</code>
<code>()</code>	Match any of the choices of string within the parentheses. So <code>(this that)</code> will match either 'this' or 'that', but not both
<code>^</code>	Anchor the pattern to the start of the line. So <code>^The</code> will match only lines beginning with <code>The</code>
<code>\$</code>	Anchor the pattern to the end of the line (placed directly after the pattern). So <code>\.\$</code> will match lines ending in a full stop
<code>\b</code>	Word boundary marker. Placed before and after an expression to ensure that only instances of full words are searched. So <code>\bshop\b</code> matches 'shop' when it appears as a complete word, but not when part of another word, such as 'shopper'
<code>\s</code>	Match a space, including tabs. So <code>a\s b</code> will match 'a b'
<code>\n</code>	Match a new line. So <code>\n</code> will match lines ending in a full stop
<code>\t</code>	Match a tab character

Next month

SET UP A WEB PROXY

Configure Squid to speed up your connection